MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Strategic Planning with Artificial Intelligence for Agile Coordination of an Industrial Robot Cell

Students' Scientific Conference papers

Author
Levente Vajda

Department Supervisor
Dr. Attila Vidács
Associate Professor

Industrial Supervisor
Dr. Géza Szabó
Ericsson Ltd.

October 23, 2019

Contents

Kivonat **3**

Abstract **4**

1 Introduction **5**

2 Related work **7**

3 Used software components **9**

3.1 Robot Operating System . . . 9

3.1.1 ROS Concepts . . . 10

3.1.2 ROS Computation Graph level . . . 11

3.1.3 Summary . . . 14

3.2 Used ROS packages . . . 15

3.2.1 ariac package . . . 15

3.2.2 rosplan package . . . 15

3.3 Gazebo . . . 15

3.3.1 Physics engine . . . 17

3.3.2 What is a simulated World? . . . 18

3.3.3 Main components . . . 19

3.3.4 Using Gazebo with ROS . . . 20

3.3.5 UR10 Robot arm . . . 21

3.3.6 Summary . . . 21

3.4 Automated planning and scheduling . . . 21

3.4.1 Overview . . . 21

3.4.2 The language of planning problems . . . 22

3.4.3 Advantages of Planning . . . 23

4 Implementation **25**

4.1 Creating domain and problem files . . . 25

4.1.1 Domain file . . . 25

4.1.2 Problem file 28
4.1.3 Solution file 31
4.2 Planner 31
4.2.1 LPG-td 31
4.3 Changing the simulated environment 33
4.3.1 Changing the behaviour of the conveyor belt 33
4.3.2 Modification of the box model file 33
4.3.3 Installation of sensors 34
4.4 Modifying and using ROSPlan 34
4.4.1 Creating low-level actions 35

5 Results and evaluation **37**
5.1 Issues with planning 37
5.2 Visualization of the plan 38
5.3 Advantages of the simulation 39
5.4 Executing the solution 41

6 Conclusion and further work **43**

Bibliography **44**

Kivonat

A gyárakban használt ipari robotok többsége előre beprogramozott cselekvéssorozatot hajt végre. A robotoknak nincs ismeretük a körülöttük lévő világról. Egy esetleges hiba esetén az egész robotcella és a gyártás is leáll. A robotok nem tudnak eltérni a beprogramozott koreográfiától, ezért nem is képesek maguk megoldani a felmerülő problémákat, valamint adaptálódni a változásokhoz. A nem tervezett leállás a legrosszabb, például egy gépjárműgyártó esetén, ahol percenként egy autó elhagyja a gyártósort. Egy leállás percenként 20 000 dollár veszteséget is eredményezhet, és egyetlen incidens akár 2 millió dollárba kerülhet. [1]

A kiinduló kérdésem az volt, hogy hogyan lehetne ipari feladatokat úgy megoldani, hogy nem megegyező lépések állandó sorozatát valósítjuk meg, hanem az éppen adott problémához igazodva adunk egy optimális tervet. A munkám célja egy olyan rendszer bemutatása, amely Mesterséges Intelligencia (MI) felhasználásával készít egy tervet a környezeti változókat is figyelembe véve. Ehhez Gazebo szimulációs szoftvert használok, amellyel hatékonyan lehet szimulálni komplex környezeteket, valamint Robot Operating Systemet (ROS), amely egy robot alkalmazásfejlesztő platform. A probléma megoldására tervkészítést alkalmazok, amely a MI régóta létező részterülete.

Bemutatok egy szimulált környezetet, melyben a robotkarnak az utasításokat egy MI adja. A feladat, hogy alkatrészeket tegyen be pontosan egy futószalagon lévő tárolóba. A tervet mindig az adott környezeti állapotot figyelembe véve készíti el az MI.

Abstract

Most industrial robots used in factories perform a pre-programmed sequence of actions. The robots have no knowledge of the world around them. In the event of a malfunction, the entire robot cell and the production are shut down. Robots cannot deviate from programmed choreography, so they cannot solve problems themselves and adapt to changes. Unplanned downtime is a worst-case scenario for automotive manufacturers, where one car leaves the production line every minute. The downtime can cost as much as $20,000 potential profit loss per minute, and $2 million for a single incident.[1]

My initial question was how to solve industrial problems without implementing a constant series of identical steps, but to provide an optimal plan based on the problem at hand.

The purpose of my work is to introduce a system that uses Artificial Intelligence (AI) to make a plan that takes into account environmental variables. My analysis is performed on a robot simulation software, Gazebo interacting with ROS. Gazebo is an open source simulation software that can be used to simulate complex environments. Robot Operating System (ROS) is a robotic application development platform. To solve problems, I used planning, which is a well-established sub-area of artificial intelligence.

In this paper, I present a simulated environment in which the robot arm is instructed by an AI. The task that needs to be solved is to place the parts precisely into a shipping box on the conveyor belt. The AI computes the plan according to the given environmental condition.

Chapter 1

Introduction

The robots used in factories always carry out a pre-programmed task for assembling, welding, gluing and palletizing. This is not a problem as long as everything goes as planned, but any changes, such as moving a part a little further away, can cause serious problems. Because the robot is not aware of its environment, it cannot adapt to the changes. Today, two-thirds of automotive workers-the human ones-are in the general assembly section. Automating this section has proved more difficult because the customization and complexity of today's autos requires the flexibility humans provide. Most factories are producing several models of cars simultaneously, and the mix of those models is often changing depending on demand. It would be expensive, if even possible, to reprogram robots and machines to be able to accommodate daily changes in factory production schedules. [2] In this paper I would suggest a possible solution to the problems mentioned above.

In 2018 and 2019 I participated in the Agile Robotics for Industrial Automation Competition (ARIAC)[5] which is a simulation-based competition designed to promote agility in industrial robotics using the latest developments in artificial intelligence and robot planning. The general goal of the first, second and third editions (2017, 2018, 2019) of the ARIAC competition were to motivate further development and adoption of agile industrial robotics by providing an environment where teams could work on solutions towards more productive and autonomous robots that would also require less time from shop floor workers. But no one used a higher level planner to solve the competition tasks, everyone created some sort of scheduler that was able to solve the problem, but optimal solution was not guaranteed.

The competition involved a simulation of the infrastructure where teams would have to complete a set of tasks. The simulation infrastructure was built on Gazebo [4] and ROS [6]. The tasks were made to comprehend four specific areas: failure identification and recovery, automated planning, fixtureless environment, and plug and play robots. The tasks or challenges were explored with different simulation trials, which represent the configuration of the simulated environment as well as its goals. ARIAC tasks revolve around collecting a set of part pieces and placing them on a tray to be sent for assembling.

Recent advances in simulator technology go beyond process level simulation e.g., [3] and with the application of rigid body simulation, a detailed, close-to-real world implementation study can be performed. I chose Gazebo as my target robot simulation environment. Gazebo [4] offers the ability to efficiently and accurately simulate a great number of robots in complex indoor and outdoor environments. It has a robust physics engine, convenient programmatic and graphical interfaces and high-quality graphics. Gazebo is free and widely used among robotic experts. The physics engine is used to model the behaviors of objects in space. These engines allow the simulation of different types of bodies to be affected by various physical stimuli. There are two types of physics engines: real-time and the high precision. Most real-time engines are inaccurate and only provide reduced approximation of the real world, while most high-precision engines are too slow for everyday applications. Physics engines are based on the laws of classical mechanics. The used models determine how accurate these simulations are in dynamical simulations. Gazebo uses the later, sacrificing performance over accuracy, which can be fine-tuned by several parameters.

A rigid body simulator would be difficult to apply for the evaluation of complex robotic cell task, but due to a robotic competitions, Gazebo starts to gain new features to support this.

In this paper, I would like to present a solution in the 2018 ARIAC environment, my goal was to make the robot cell more adaptive, versatile, and optimized. To do this, I used the features offered by AI planning, which is a long-established branch of research in Artificial Intelligence. About AI planning, Gazebo and ROS I provide background information in chapter 3 to better understand their role in my work. In chapter 4, I will outline the steps that I had to take in order to create such a system using the previously mentioned components. Then in chapter 5 I present my results and experiences. Finally, I will discuss other possible applications and the potential continuation of my work.

Chapter 2

Related work

Many autonomous systems such as mobile robots, UAVs or spacecraft, have limited resource capacities and move in dynamic environments. Performing on-board mission planning and execution in such a context requires deliberative capabilities to generate plans achieving mission goals while respecting deadlines and resource constraints, as well as runtime plan adaption mechanisms during execution. Authors of [7] propose a framework to integrate deliberative planning, plan repair and execution control in a dynamic environment with stringent temporal constraints. It is based on lifted partial order temporal planning techniques which produce flexible plans and allow, under certain conditions discussed in the paper, plan repair interleaved with plan execution. This framework has been implemented using the IXTET planner and used to control a robotic platform

Authors of [8] explore the execution of planned AUV missions where opportunities to achieve additional utility can arise during execution. The missions are represented as temporal planning problems, with hard goals and time constraints. Opportunities are soft goals with high utility. The probability distributions for the occurrences of these opportunities are not known, but it is known that they are unlikely so it is not worth trying to anticipate their occurrence prior to plan execution. However, as they are high utility, it is worth trying to address them dynamically when they are encountered, as long as this can be done without sacrificing the achievement of the hard goals of the problem. They formally characterise the opportunistic planning problem, introduce a novel approach to opportunistic planning and compare it to an on-board replanning approach in the domain of autonomous underwater vehicles performing pillar exception and chain following tasks. Authors of [9] present technology for performing autonomous commanding of a planetary rover. Through the use of AI planning, scheduling and execution techniques, the OASIS autonomous science system provides capabilities for the automated generation of a rover activity plan based on science priorities, the handling of opportunistic science, including new science targets identified by onboard data analysis software, other dynamic decision-making such as modifying the rover activity plan in response to problems or other state and resource changes. We first describe some of the particular challenges this work has begun to address and then describe our system approach. Finally, we report on our experience testing this software with a Mars rover prototype.

Authors of [10] describe the Remote Agent flight experiment for spacecraft commanding and control. In the Remote Agent approach, the operational rules and constraints are encoded in the flight software. The software may be considered to be an autonomous "remote agent" of the spacecraft operators in the sense that the operators rely on the agent to achieve particular goals.

[11] is a survey of articles and methods on how to optimize a robot cell. The purpose of such optimization is to minimize or maximize at least one of the following objective functions: 1) minimizing the execution time, respectively maximizing the robot productivity, considering that the relative speeds of the actuators elements are limited constructively; 2) minimizing the energy consumption or mechanical work necessary for execution, leading to a reduction of the mechanical stresses in actuators and on the robot structure and obtaining smooth trajectories, easy to follow; 3) minimizing the maximum power required for operating the robot; 4) minimizing the maximum actuation forces and moments. The most common optimization criteria used in the literature are: minimum time trajectory planning; minimum energy trajectory planning or minimum actuation effort and minimum jerk trajectory planning.

Authors of [12] highlighted key conclusions from a workshop sponsored by the National Science Foundation in October 2013 that summarize opportunities and key challenges in robot planning and include challenge problems identified in the workshop that can help guide future research toward making robot planning more deployable in the real world. This article highlights that robot planning could also be used in nimble factories with rapidly changing products and needs by facilitating quick adaptation to new tasks and reducing the effort of manually reprogramming robots if workspaces or products are modified. Creating robots with the planning capabilities needed for these new scenarios will require research on manipulation planning, efficient user interfaces for conveying how tasks should be performed, human-robot cooperation, enabling situational awareness, compensating for environmental and operational uncertainty, and assuring performance. Although robots are increasingly being used in a variety of real-world applications, the deployment of advanced robot planning capabilities in real-world robots has thus far been limited. Progress will require the collaboration of planning researchers from robotics and artificial intelligence with researchers from neighboring disciplines, such as computer vision, haptics, natural language processing, and human-computer interfaces.

The papers above outline the many possibilities of using AI planning from space to deep sea. They mention that many components are required to be used in real-world scenarios. I present a possible solution to these issues in this paper.

Chapter 3

Used software components

In this chapter, I present the basics of the Robot Operating System (ROS), Gazebo simulation software and planning with PDDL, with the aim of presenting their role in my work.

3.1 Robot Operating System

Robot Operating System (ROS)[13] is a trending robot application development platform that provides various features such as message passing, distributed computing, code reusing, and so on.

ROS comes with ready to use capabilities, for example, SLAM (Simultaneous Localization and Mapping) and AMCL (Adaptive Monte Carlo Localization) packages in ROS can be used for performing autonomous navigation in mobile robots and the MoveIt package for motion planning of robot manipulators. These capabilities can directly be used in our robot software without any hassle. These capabilities are its best form of implementation, so writing new code for existing capabilities are like reinventing wheels. Also, these capabilities are highly configurable; we can fine-tune each capability using various parameters.

ROS is packed with tons of tools for debugging, visualizing, and performing simulation. The tools such as rqt_gui, RViz and Gazebo are some of the strong open source tools for debugging, visualization, and simulation. The software framework that has these many tools is very rare.

The ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has contributed support for Fedora, Gentoo, Arch Linux and other Linux platforms. [6, 14]

3.1.1 ROS Concepts

ROS has three levels of concepts: the File system level, the Computation Graph level, and the Community level. [15] In the next chapters I will show the most important elements of these three levels.

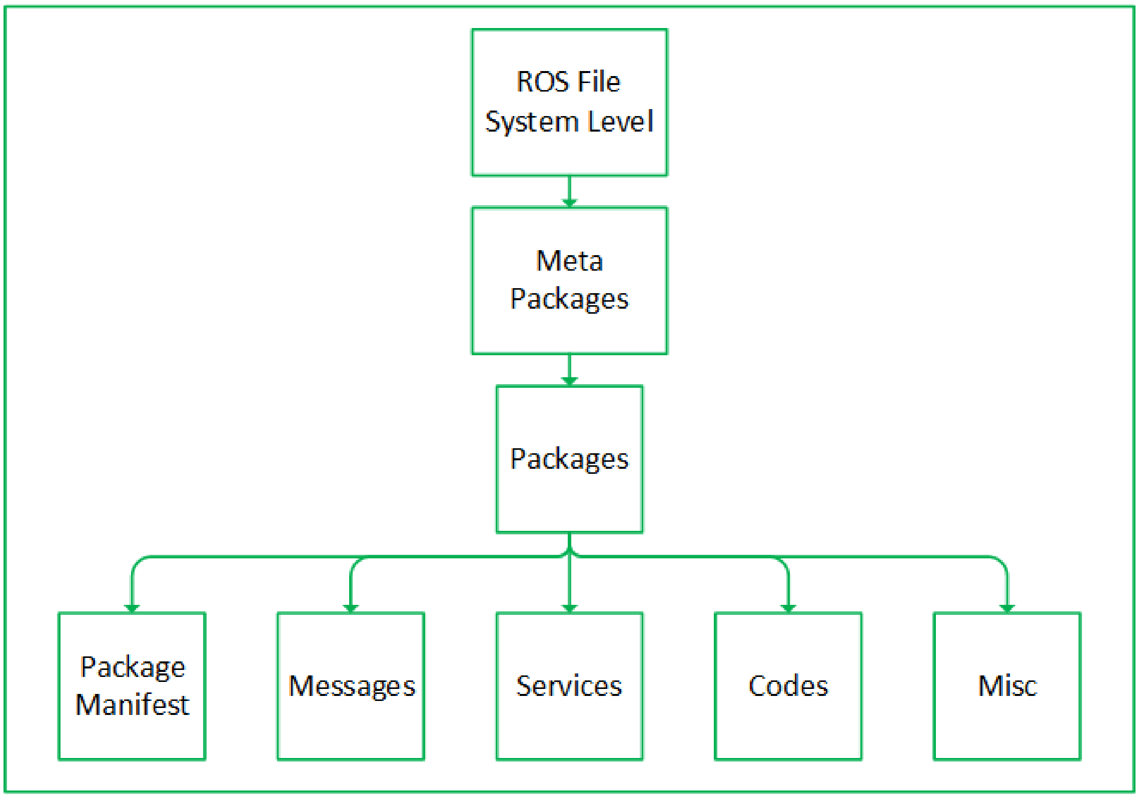


Figure 3.1: *ROS File system level[13]*

3.1.1.1 Packages

Software in ROS is organized in packages[16]. A package might contain a library, a dataset, configuration files or anything else that logically constitutes a useful module. The goal of these packages it to provide this functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow the "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

3.1.2 ROS Computation Graph level

The computational graph of ROS is peer-to-peer network that processes data. The elements of the computational graph are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.[15]

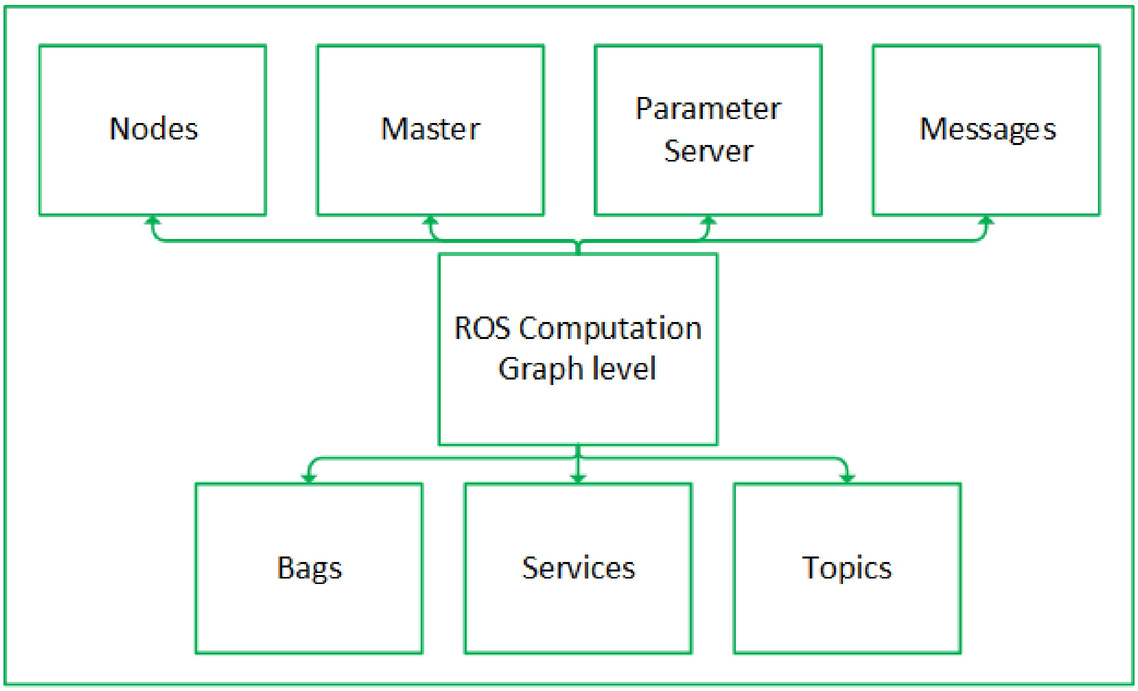


Figure 3.2: *ROS Computation Graph level[13]*

3.1.2.1 Nodes

ROS nodes[17] are responsible for performing tasks that require computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server.

Using ROS nodes makes the architecture of the systems more transparent, as each node performs a single function. Nodes provide more stable system, because in case of an error, if a node fails, the whole system does not stop. Code complexity is reduced in comparison to monolithic systems. In the case of a system change, if a function is replaced by a better one, it is not necessary to change the entire system, it is enough to replace a node with another one. This interchangeability is also present in the ROS framework, for example, it is not necessary for nodes to be written in the same programming language in order to communicate. ROS nodes can be written in python and C++.

Every node has a URI[18], which corresponds to the host:port of the XMLRPC server it is running. The XMLRPC server is not used to transport topic or service data: instead, it is used to negotiate connections with other nodes and also communicate with the Master.

This server is created and managed within the ROS client library, but is generally not visible to the client library user. The XMLRPC server may be bound to any port on the host where the node is running.

3.1.2.2 Master

The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.[15] It tracks publishers and subscribers to topics as well as services. The role of the Master[19] is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. The Master also provides the Parameter Server.

The Master[18] is implemented via XMLRPC , which is a stateless, HTTP-based protocol. XMLRPC is relatively lightweight, does not require a stateful connection, and has wide availability in a variety of programming languages, for example Perl, Java, Python, C, C++, PHP.

3.1.2.3 Parameter Server

Parameter server[13] is a shared database between nodes. It can stores globally accessible variables that every node can read, write or delete, but it is also possible to control access to the variables. The server can store a wide variety of data types, for example 32-bit integers, booleans, strings, doubles, iso8601 dates and can even store dictionaries. If the number of parameters is too high it is possible to save them in a YAML Ain't Markup Language (YAML) file.

The Parameter Server[18] is part of the Master, which means that its implemented via XMLRPC libraries in order to enable easy integration with the ROS client libraries and also to provide greater type flexibility when stroring and retrieving data.

3.1.2.4 Messages

ROS nodes communicate with each other by publishing messages to a topic. It supports standard primitive data types[13] (e.g., integer, floating point, Boolean, and so on) and arrays of primitive data types. It is also possible to build our own message types using these standard types.

Messages[20] files are simple text files with .msg extension stored in the msg subdirectory of a package. Message generators translate the .msg files into source code. These message generators are implemented in the Client Libraries and they are invoked from build script.

The are two commonly used message packages std_msgs and common_msgs. The messages in std_msgs[21] package are not intended for permanent usage because these types do not convey sematic meaning about ther contents. Every message simply has a field called "data". The messages in this package can be useful for quick prototyping, for example the Empty type, which is useful for sending an empty signal or the "MultiArray" types, which can be useful for storing sensor data.

Messages in the common_msgs[22] package are widely used by other ROS packages. It is being used to interact with an action server and an action client in the actionlib_msgs, for diagnostics and runtime monitoring in diagnostic_msgs, for common geometric primitives such as points, vectors, and poses in geometry_msgs, for robot navigation in nav_msgs and common sensors (sensor_msgs), such as laser range finders, cameras, point clouds.

3.1.2.5 Topics

Topics[23] are named buses over which nodes exchange messages. Topics have anonymous publish and subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining small amounts of state.

Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol.[15] Two transport protocols are supported by ROS[18], one based on TCP/IP, another UDP-based. TCP is widely used because it provides a simple, reliable communication stream. TCP packets always arrive in order, and lost packets are resent until they arrive. The TCP/IP-based transport is called as TCPROS and it is the default protocol used in ROS. While TCP is great for wired Ethernet networks, these features become bugs when the underlying network is a lossy WiFi or cell modem connection. In this situation, UDP is more appropriate. When multiple subscribers are grouped on a single subnet, it may be most efficient for the publisher to communicate with all of them simultaneously via UDP broadcast. The UDP-based protocol is known as UDPROS. For these reasons, ROS does not commit to a single transport. Given a publisher URI, a subscribing node negotiates a connection, using the appropriate transport, with that publisher, via XMLRPC. The result of the negotiation is that the two nodes are connected, with messages streaming from publisher to subscriber.

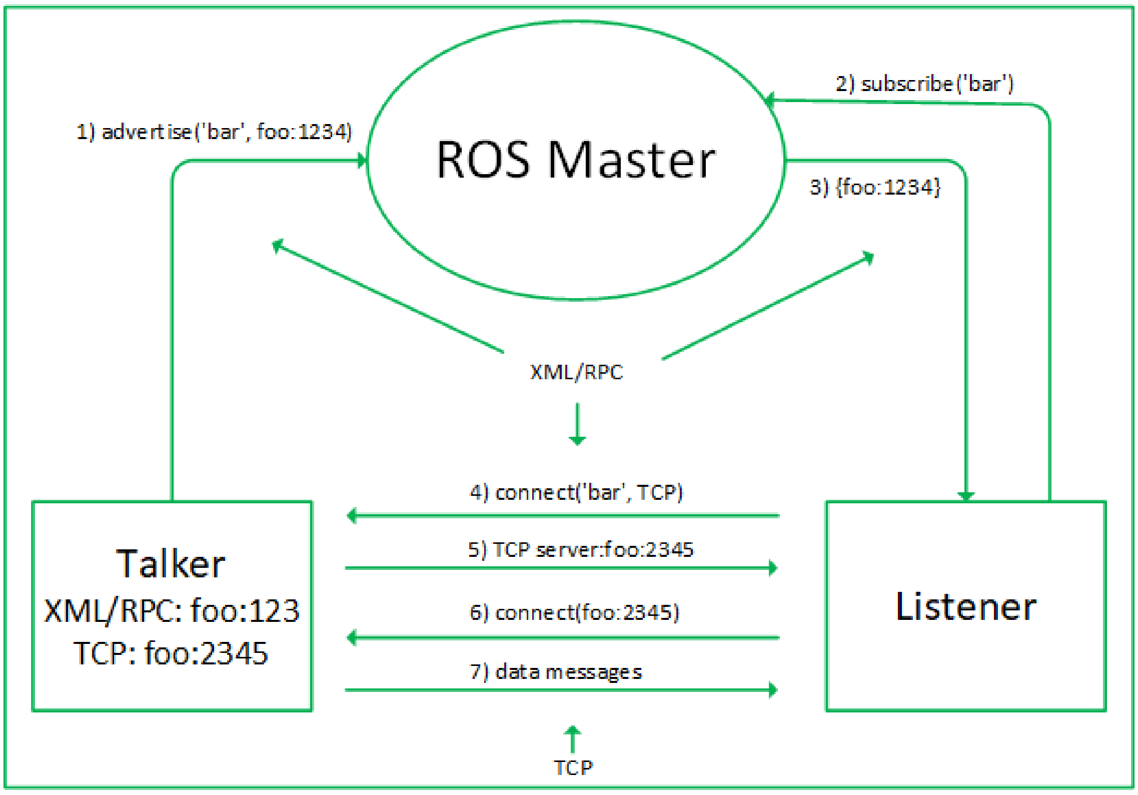


Figure 3.3: *ROS communication between two nodes and the master[18]*

3.1.2.6 Services

Request/reply interaction can only be achieved with ROS services[13]. Topics can not do this kind of communication because it is unidirectional. ROS services are mainly used in a distributed system. The ROS services are defined using a pair of messages. We have to define a request datatype and a response datatype in a .srv file. The .srv files are kept in a srv folder inside a package. In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client.

3.1.3 Summary

ROS is an open-source framework to develop robotic applications. Because of the structure that each package has a separate function, it makes it considerably easier for developers to work, since there is no need for reinventing the wheel.

Thanks to a very active community of helpful people, if you are facing a problem you will most likely find people who have been in a similar shoe.

In my opinion, ROS is a good choice for my work because, although at the beginning there is a big enough threshold to jump. It is not easy to learn how to use it, but once one has overcome the initial difficulties, it is easier to create complex, scalable, transparent systems.

3.2 Used ROS packages

To avoid having to deal with something that has already been made, I have used the following packages to speed up and ease my work.

3.2.1 ariac package

The ariac package that I am worked with was made for the 2018 ARIAC. The ariac package contains the simulation environment and the GEAR interface. GEAR[24] provides a ROS interface to control all available actuators, read sensor information and send/receive notifications. The ariac package is built on several other packages, but I will not go into detail about them.

3.2.2 rosplan package

The ROSPlan[25] framework provides a collection of tools for AI Planning in a ROS system. ROSPlan has a variety of nodes which encapsulate planning, problem generation, and plan execution. This structure is shown in 3.4 figure.

The Knowledge Base[25] stores the PDDL model. It stores both a domain model and the current problem instance. The Problem Interface node is used to generate a problem instance. It fetches the domain details and current state through service calls to a Knowledge Base node and publishes a PDDL problem instance as a string, or writes it to file. The Planner Interface node is a wrapper for the AI Planner. The planner is called through a service, which returns true if a solution was found. This interface feeds the planner with a domain file and problem instance, and calls the planner with a command line specified by parameter.The Parsing Interface node is used to convert planner output into a plan representation that can be executed, and whose actions can be dispatched to other parts of the system. Plan Dispatch includes plan execution, and the process of connecting single actions to the processes which are responsible for their execution. An implementation of the Plan Dispatch node subscribes to a plan topic, and is closely tied to the plan representation of plans published on that topic.

3.3 Gazebo

Gazebo[4] is a 3D dynamic Simulator that can simulate accurately and efficiently populations of robots in complex indoor or outdoor environments. This way, we can easily

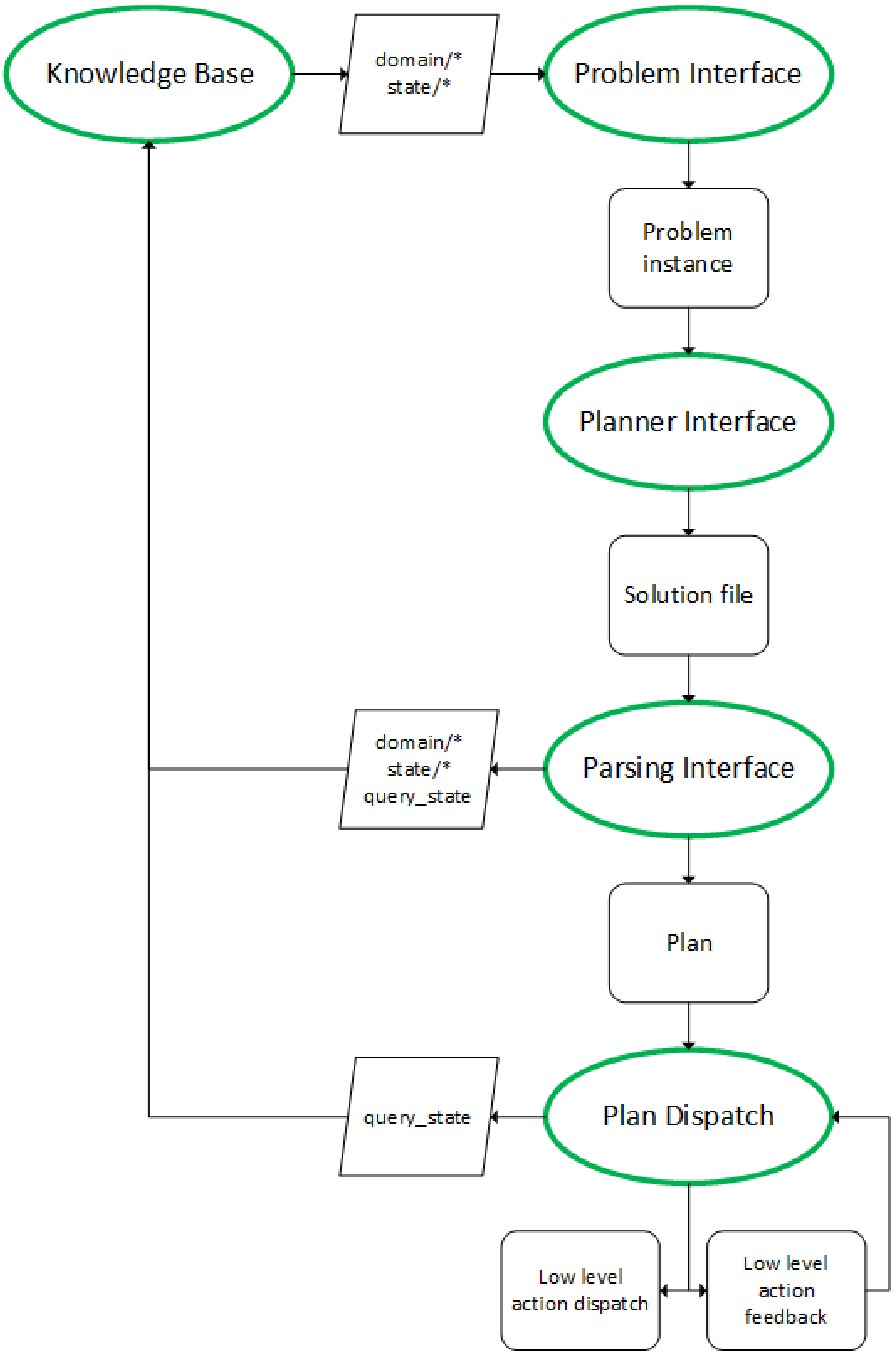
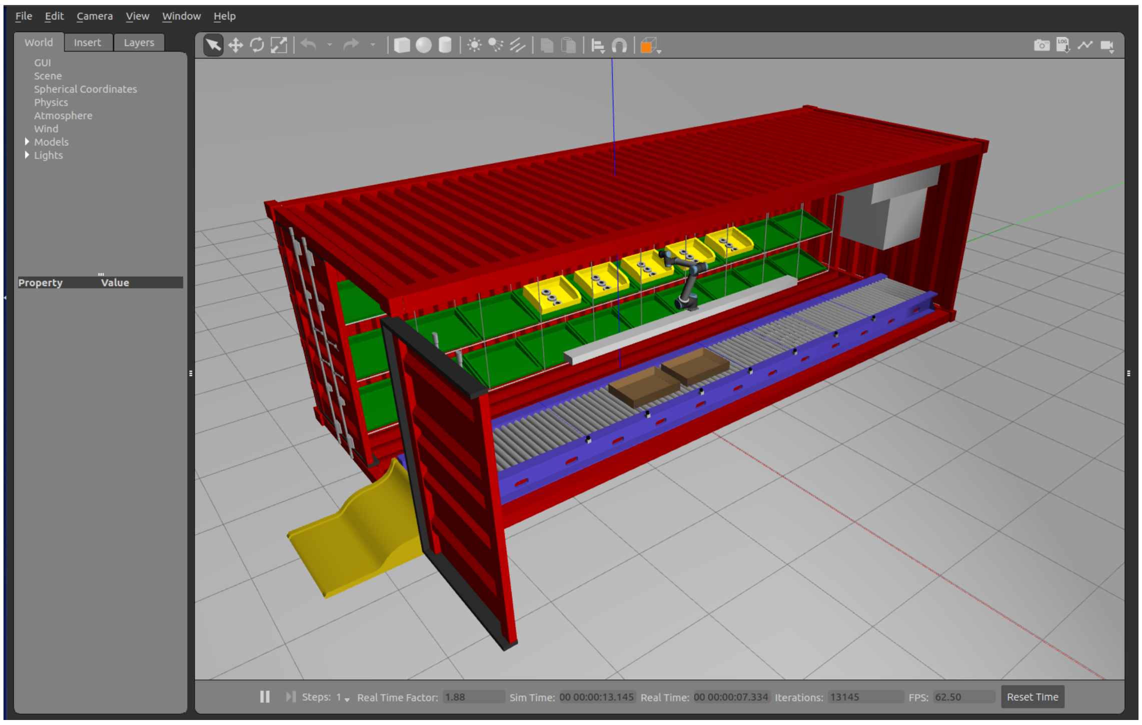


Figure 3.4: *The structure of the ROSPlan framework, the package can be divided into 5 separate nodes whose relationship to each other is shown in the figure.*

test robotic applications and algorithms without the need for actual hardware. Gazebo provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.



Steps: 1 Real Time Factor: 1.88 Sim Time: 00 00:00:13.145 Real Time: 00 00:00:07.334 Iterations: 13145 FPS: 62.50

Figure 3.5: *Gazebo simulation of the 2018 ARIAC environment*

3.3.1 Physics engine

Gazebo[4] supports the ODE, Bullet, Simbody and DART physics engines. By default Gazebo is compiled with support for ODE. In my work I used only the Open Dynamics Engine (ODE), therefore I would only give a more detailed description of that physics engine.

Open Dynamics Engine (ODE)[26] is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

The process of simulating the rigid body system through time is called integration.[27] Each integration step advances the current time by a given step size, adjusting the state of all the rigid bodies for the new time value. There are two main issues to consider when working with any integrator:

- How accurate is it? That is, how closely does the behaviour of the simulated system match what would happen in real life.
- How stable is it? That is, will calculation errors ever cause completely non-physical behaviour of the simulated system? (e.g. causing the system to "explode" for no reason).

The current integrator of ODE is very stable, but not particularly accurate unless the step size is small. In gazebo the default step size is 0.001, which means it calls the UpdatePhysics resource 1000 times in a simulated second. With this setting, the simulation will be very realistic so the events in the simulation are a good approximation of reality. Unfortunately, this means that if we want to simulate too complex system on a computer that has a finite computing capacity, the simulation speed will be significantly reduced.

Gazebo has a metric that shows how slow the simulation is for real time. This is the real time factor (RTF) which is the simulation time divided by real time.

3.3.2 What is a simulated World?

A complete environment is essentially a collection of models and sensors.[28] The ground and buildings represent stationary models while robots and other objects are dynamic. Sensors remain separate from the dynamic simulation since they only collect data, or emit data if it is an active sensor

The following is a brief description of each general component[28]:

- Models, Bodies, and Joints: A model is any object that maintains a physical representation. This encompasses anything from simple geometry to complex robots. Models are composed of at least one rigid body, zero or more joints and sensors, and interfaces to facilitate the flow of data. Bodies represent the basic building blocks of a model. Their physical representation take the form of geometric shapes chosen from boxes, spheres, cylinders, planes, and lines but it is also possible to use complex model files. Each body has an assigned mass, friction, bounce factor, and rendering properties such as color, texture, transparency, etc.
- Joints provide the mechanism to connect bodies together to form kinematic and dynamic relationships. A variety of joints are available including hinge joints for rotation along one or two axis, slider joints for translation along a single axis, ball and socket joints, and universal joints for rotation about two perpendicular joints. Besides connecting two bodies together, these joints can act like motors. When a force is applied to a joint, the friction between the connected body and other bodies cause motion. However, special care needs to be taken when connecting many joints

in a single model as both the model and simulation can easily loose stability if incorrect parameters are chosen.

- Interfaces provide the means by which client programs can access and control models. Commands sent over an interface can instruct a model to move joints, change the configuration of associated sensors, or request sensor data. The interfaces do not place restrictions on a model, thereby allowing the model to interpret the commands in anyway it sees fit.

- Sensors: A robot can not perform useful tasks without sensors. A sensor in Gazebo is an abstract device lacking a physical representation. It only gains embodiment when incorporated into a model. This feature allows for the reuse of sensors in numerous models thereby reducing code and confusion.

3.3.3 Main components

Gazebo is a stand-alone software, this section describes each of the items involved in running a Gazebo simulation. [29]

- World file: The world description file contains all the elements in a simulation, including robots, lights, sensors, and static objects. This file is formatted using Simulation Description Format (SDF), and typically has a .world extension.

- Model files: A model file uses the same SDF format as world files, but should only contain a single model tag. We can also use URDF files to create models, because we can convert the URDF file into an SDF. The purpose of these files is to facilitate model reuse, and simplify world files. Once a model file is created, it can be included in a world file.

- Gazebo Server: The server is the workhorse of Gazebo. It parses a world description file given on the command line, and then simulates the world using a physics and sensor engine.

- Graphical Client: The graphical client connects to a running gzserver and visualizes the elements. This is also a tool which allows you to modify the running simulation.

- Plugins: Plugins provide a simple and convenient mechanism to interface with Gazebo. Plugins can either be loaded on the command line, or specified in an SDF file.

3.3.4 Using Gazebo with ROS

Gazebo is a stand-alone simulator. However Gazebo has an interface, through which we can intervene into the simulation with ROS.

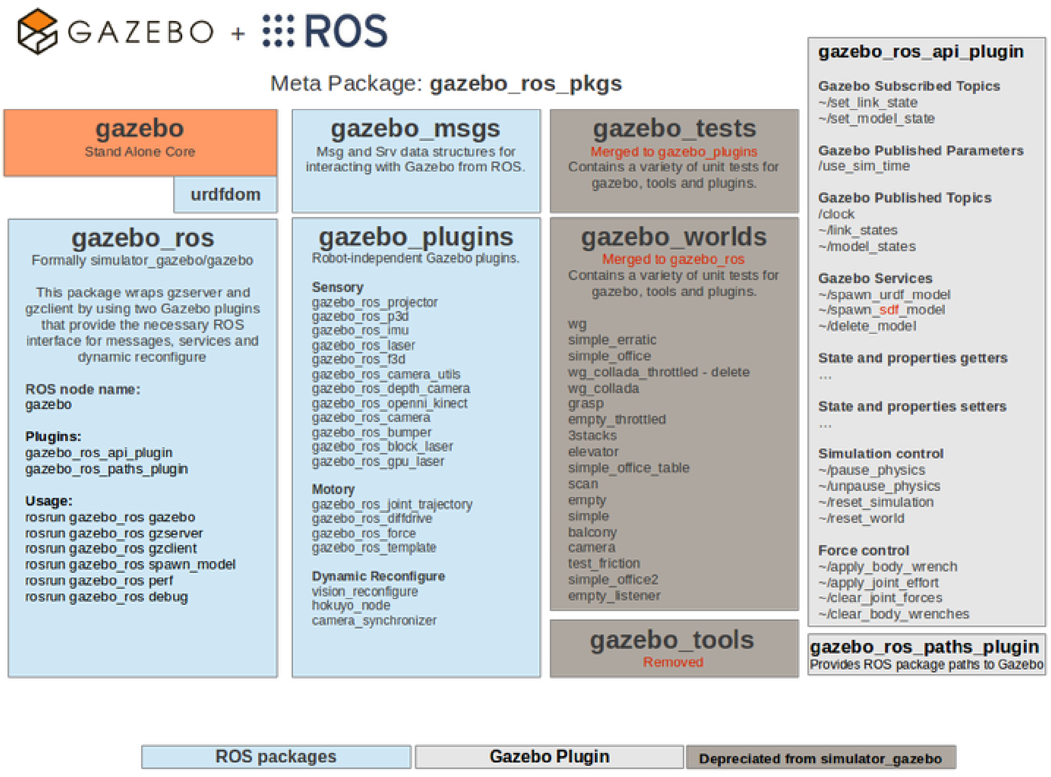


Figure 3.6: *An overview of the gazebo_ros_pckgs interface [30]*

3.3.4.1 gazebo_ros_pkgs package

gazebo_ros_pkgs [30] provides wrappers around the Gazebo to achieve ROS integration. This is a meta package that contains the necessary interfaces to simulate a robot in Gazebo using ROS messages, services and dynamic reconfigure.

The use of each package is as folows [13]:

- gazebo_ros: This contains wrappers and tools for interfacing ROS with Gazebo
- gazebo-msgs: This contains messages and service data structures for interfacing with Gazebo from ROS
- gazebo-plugins: This contains Gazebo plugins for sensors, actuators, and so on.

- gazebo-ros-control: This contains standard controllers to communicate between ROS and Gazebo

3.3.5 UR10 Robot arm

In the ARIAC environment, there is a UR10 robot arm that is responsible for moving parts from the yellow boxes. It also plays a significant role in my work, so I present its essential characteristics. The UR10 robot arm designed by Universal Robots has 6 degrees of freedom with its 6 rotating joints. Its payload can be up to 10 kg. It has a reach of 1300 mm. [31]

3.3.6 Summary

Gazebo is an open-source dynamic simulator that can accurately and effectively simulate complex environments with multiple robots. Although this precision comes with a trade-off, since every interaction is computed 1000 times per second, so if the system is too complex, the simulation can slow down considerably. This is one aspect that needs to be taken into account during development.

Gazebo can be connected to ROS so the events in the simulation can be changed with ROS. This and the community around the software, as well as the resources that were available on the Internet, seemed to be an appropriate choice for my work.

3.4 Automated planning and scheduling

Automated planning and scheduling, or more simply denoted as AI planning is a well-established sub-area of artificial intelligence. The purpose of planning is for the planner to create a series of actions that meet the goal conditions.

3.4.1 Overview

From a computer science point of view, planning is just one formalism succinctly describing large transition systems, similar to automata networks or Turing machines. Trivially, planning is hard (PSPACE-complete in our case here). [32]

A planning task is consists of four parts [32]:

- a set of finite-domain state variables
- a set of actions, where each action has a partial variable assignments called preconditions and effects.
- a complete variable assignment called initial state

- a partial variable assignment called goal

A plan is path leading from the initial state to a state complying with the goal. The plan is optimal if it is a shortest such path. Unless other metrics are considered during the planning. [32]

The classic planning environments that are fully observable, deterministic, finite, static (that is, where changes only occur when the agent acts), discrete (in time, actions, objects and effects). This is a fairly significant limitation, so most problems cannot be solved with classic planning. Non-classical planning deals with partially observable, stochastic, dynamic environments and with multiple agents. [33]

3.4.2 The language of planning problems

In the late 1990s, the yearly International Planning Competitions (IPL) began in order to encourage development of efficient planning search algorithms. A new language, PDDL (Planning Domain Definition Language) was designed in order to have a single unified syntax for representing planning problems for the competition. PDDL has improved over time to support many sophisticated features, such as variable types, action costs, action preferences, deadlines, etc. even though no planning software available today supports all these features of PDDL. [34]

The main inspiration for the PDDL language was the STRIPS language from the early 1970s, developed at SRI International. STRIPS (Stanford Research Institute (SRI) Problem Solver) was used to help Shakey the Robot solve various tasks. [34]

Planning problems represented in PDDL are separated into two parts: the domain and one or more problems. The domain PDDL file lists the actions available in the domain. A problem PDDL file describes the initial state and goal criteria for a particular scenario. The idea is that the same domain file will be useful for many different problem scenarios. [34]

A domain file, typically named domain.pddl, has the following sections: [34]

- the name of the domain
- a list of predicates and each predicate's variables (recall that a predicate represents a single property of a state)
- a list of actions; for each action, its parameters, preconditions, and effects may be stated

Here is an example domain file, for the "blocks world" planning problem. Note that variables are marked with a question mark, as in ?x, and the syntax generally uses Lisp's prefix notation and keywords (preceded by colons, e.g., :parameters). [34]

```
;; domain.pddl for blocksworld

(define (domain blocksworld)
(:requirements :strips)

(:predicates    (clear ?x)
                (on-table ?x)
                (holding ?x)
                (on ?x ?y))

(:action pickup
        :parameters (?ob)
        :precondition (and (clear ?ob) (on-table ?ob))
        :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))))

(:action putdown
        :parameters (?ob)
        :precondition (and (holding ?ob))
        :effect (and (clear ?ob) (on-table ?ob)
                        (not (holding ?ob))))

(:action stack
        :parameters (?ob ?underob)
        :precondition (and  (clear ?underob) (holding ?ob))
        :effect (and (clear ?ob) (on ?ob ?underob)
                        (not (clear ?underob)) (not (holding ?ob))))

(:action unstack
        :parameters (?ob ?underob)
        :precondition (and (on ?ob ?underob) (clear ?ob))
        :effect (and (holding ?ob) (clear ?underob)
                (not (on ?ob ?underob)) (not (clear ?ob)))))
```

A problem file lists the domain, objects that can be used in place of variables, a description of the initial state (using the predicates listed in the domain file), and goal criteria (again, using predicates). [34]

```
;; problem file: blocksworld-prob1.pddl

(define (problem blocksworld-prob1)
(:domain blocksworld)
(:objects a b)
(:init (on-table a) (on-table b) (clear a) (clear b))
(:goal (and (on a b))))
```

3.4.3 Advantages of Planning

What characterizes planning research is the attempt to create one planning solver that will perform sufficiently well on all possible domains (inputs). That will never work out (the problem is hard), but there's been tremendous algorithmic progress in the last decade. This will pay off if either: [32]

- The problem is subject to frequent change. If a custom solver is implemented, it needs to be continuously adapted. Using planning, it suffices to change the declarative planning model.
- It would be costly to implement a custom solver. Unless the problem is quite easy, making a solver will cost time and money. Writing the planning model is typically much less effort.

AI planning has several important benefits that make it stand out from other AI subareas [35]:

- When explainability is desired, if we want to be able to explain why a particular course of action was chosen e.g. ass
- Rapid prototyping: short time to solution.
- Variety of off the shelf planners available both proprietary and open-source

In other words, planning is a cost-effective method for software engineering. It's a model-based approach. The planning model serves as a high-level programming language decoupling the problem from its solution.

Chapter 4

Implementation

4.1 Creating domain and problem files

In this section, I provide a brief description of what factors I took into account when I created the domain and problem files, and why I implemented it in this way.

4.1.1 Domain file

The domain file must include all actions that can change the state of the world. In addition to the changes we make, we also need to write down the conditions under which the action is executed. It is important to have a good knowledge of the environment when creating a domain file, because we do not want to put unnecessary restrictions in it, but we also do not want to create actions that are physically impossible.

The 2018 ARIAC environment is suitably limited to avoid a state space explosion. Two agents can change the state of the world, the robot arm and the conveyor belt. Therefore, 4 actions are sufficient.

The first action is responsible for moving the robot arm parallel to the conveyor belt.

```
    (:durative-action ur_move
:parameters (?ur - ur ?from - yellowbox ?to - yellowbox)

:duration (= ?duration (+ (+
(*(-(abs_X_position ?from) (abs_X_position ?to)) (-(abs_X_position ?from) (
    ↪ abs_X_position ?to)))
(*(-(abs_Y_position ?from) (abs_Y_position ?to)) (-(abs_Y_position ?from) (
    ↪ abs_Y_position ?to)))
) 1) )

:condition
                (and
                (at start (ur_at ?ur ?from))
                )
:effect
                (and
                (at end (ur_at ?ur ?to))
                (at start (not (ur_at ?ur ?from)))
                )
)
```

The PDDL version 2.1 allows the use of durative actions. It is not trivial what we choose to perform an action. If you choose a fixed time, e.g. when the action is certainly completed, the planner will ignore the distance between the yellow boxes. By doing this, an opportunity is missed to optimize the solution. The next option is to change the action duration depending on its parameters. The time of move action is similar to the Manhattan norm, except that in PDDL absolute values cannot be computed. The components are multiplied by themselves to get a positive number. Equation 4.1 shows how the duration is calculated.

$$Duration = (X_{from} - X_{to}) * (X_{from} - X_{to}) + (Y_{from} - Y_{to}) * (Y_{from} - Y_{to}) + 1 \qquad (4.1)$$

The condition is that the robot arm moves from point A to point B to initially be in point A. The effect is not surprisingly that it gets to point B and it will no longer be at point A.

The second action also belongs to the robot arm, it is responsible for picking up parts. There are two locations in the parameter list. ?locationA is the location where the robot

```
(:durative-action ur_pick
:parameters (?ur - ur ?locationA - location ?locationB - location ?piece - piece)

:duration (= ?duration (* (+ (+
(*(-(abs_X_position ?locationB) (abs_X_position ?locationA)) (-(abs_X_position ?
    ↪ locationB) (abs_X_position ?locationA)))
(*(-(abs_Y_position ?locationB) (abs_Y_position ?locationA)) (-(abs_Y_position ?
    ↪ locationB) (abs_Y_position ?locationA)))
) 1) 1.5) )

:condition
        (and
        (at start (ur_at ?ur ?locationA))
        (over all (ur_at ?ur ?locationA))
        (at start (is_ur_available ?ur))
        (over all (is_reachable ?locationA ?locationB))
        (at start (is_in ?piece ?locationB))
        )

:effect
        (and
        (at end (is_ur_carry ?ur ?piece))
        (at end (not (is_in ?piece ?locationB)))
        (at end (not (is_ur_available ?ur)))
        )
)
```

arm is located and ?locationB is where the part is located. Using this it is easy to calculate duration. The 1.5 times multiplier is needed to change the characteristics of the different robotic arms by setting a single parameter. The prerequisite for the pick action is that the robot be located where the part is reachable and remain there for the duration of the action. Also, it should be available, so the vacuum gripper should be free. The effect is quite self explaining, the robot arm is no longer available and it carries the part.

The place action is not much different from the pick, the location A is now the position of the robot and the location B where the part will be placed The is_precise predicate

```
(:durative-action ur_place
:parameters (?ur - ur ?locationA - location ?locationB - location ?piece - piece)

:duration (= ?duration (* (+ (+
(*(-(abs_X_position ?locationB) (abs_X_position ?locationA)) (-(abs_X_position ?
    ↪ locationB) (abs_X_position ?locationA)))
(*(-(abs_Y_position ?locationB) (abs_Y_position ?locationA)) (-(abs_Y_position ?
    ↪ locationB) (abs_Y_position ?locationA)))
) 1) 1.5) )

:condition
        (and
        (at start (ur_at ?ur ?locationA))
        (over all (ur_at ?ur ?locationA))
        (at start (is_ur_carry ?ur ?piece))
        (over all (is_reachable ?locationA ?locationB))
        )

:effect
        (and
        (at end (not (is_ur_carry ?ur ?piece)))
        (at end (is_in ?piece ?locationB))
        (at end (is_ur_available ?ur))
        (at end (is_precise ?piece ?locationB))
        )
)
```

expresses that the part is in its accurate position. It is possible to have a robot arm that is unable to place down accurately.

The shippingbox_move action moves two shipping boxes simultaneously on the conveyor belt. The action is very similar to ur_move. The *(at start (<= (*(*(- (yellowbox-position ?to) (yellowbox-position ?to2)) (- (yellowbox-position ?to2) (yellowbox-position ?to))) -1) 1))* expression is responsible for preventing the planner from changing the order of the left and right boxes. This expression is true because *yellowbox-position* values always increase from left to right and the *(at start (< (yellowbox-position ?to) (yellowbox-position ?to2)))* expression ensures that we can only move boxes from left to right. This limitation is needed because otherwise the planner may change the position of the shipping boxes relative to one another. For a similar reason, it is expressed that one shipping box is different from the other, otherwise it would be possible to move only one box at a time. What is also different from the ur_move action is that at the end of the action, the position of the shipping box is also set.

```
(:durative-action shippingbox_move
:parameters (?shippingbox1 - shippingbox ?shippingbox2 - shippingbox ?from -
    ↪ yellowbox ?to - yellowbox ?from2 - yellowbox ?to2 - yellowbox)

:duration (= ?duration (+ (+
(*(-(abs_X_position ?from) (abs_X_position ?to)) (-(abs_X_position ?from) (
    ↪ abs_X_position ?to)))
(*(-(abs_Y_position ?from) (abs_Y_position ?to)) (-(abs_Y_position ?from) (
    ↪ abs_Y_position ?to)))
) 1) )

:condition
        (and
        (at start (shippingbox_at ?shippingbox1 ?from))
        (at start (shippingbox_at ?shippingbox2 ?from2))
        (at start (< (yellowbox-position ?to) (yellowbox-position ?to2)))
        (at start (<= (*(*(- (yellowbox-position ?to) (yellowbox-position ?to2))
        (- (yellowbox-position ?to2) (yellowbox-position ?to))) -1) 1))
        (at start (is_different ?shippingbox1 ?shippingbox2))
        )

:effect
        (and
        (at end (shippingbox_at ?shippingbox1 ?to))
        (at end (not (shippingbox_at ?shippingbox1 ?from)))
        (at end (is_reachable ?to ?shippingbox1))
        (at end (is_reachable ?shippingbox1 ?to))
        (at end (not (is_reachable ?from ?shippingbox1)))
        (at end (not (is_reachable ?shippingbox1 ?from)))

        (at end (shippingbox_at ?shippingbox2 ?to2))
        (at end (not (shippingbox_at ?shippingbox2 ?from2)))
        (at end (is_reachable ?shippingbox2 ?to2))
        (at end (is_reachable ?to2 ?shippingbox2))
        (at end (not (is_reachable ?from2 ?shippingbox2)))
        (at end (not (is_reachable ?shippingbox2 ?from2)))

        (at end (assign (abs_X_position ?shippingbox1) (yellowbox-position ?to)))
        (at end (assign (abs_X_position ?shippingbox2) (yellowbox-position ?to2)))
        )
)
```

4.1.2 Problem file

The physical parameters, the initial state and the goal can be defined in the problem file. For the sake of simplicity, I chose the same type of part. There are 15 disks in the world, 3 in each yellow box. Right_nomansland and left_nomansland represent the empty positions next to yellowbox1 and 5.

```
(:objects
        disk1 disk2 disk3 disk4 disk5 disk6 disk7 disk8 disk9 disk10 disk11 disk12
            ↪ disk13 disk14 disk15 - disk
        ur - ur
        right_nomansland left_nomansland yellowbox1 yellowbox2 yellowbox3
            ↪ yellowbox4 yellowbox5 - yellowbox
        shippingbox1 shippingbox2 - shippingbox
)
```

When writing the initial state, the position and properties of each component must be specified, since there is a closed world assumption, everything not listed here is false by default. It is worth noting that it is important for the planner that if location B is available from location A, then it must be stated that location B is also available from A.

```
(:init
        (ur_at ur yellowbox3)
        (is_ur_available ur)

        (is_different shippingbox1 shippingbox2)
        (shippingbox_at shippingbox1 yellowbox1)
        (shippingbox_at shippingbox2 yellowbox2)
        (is_reachable shippingbox1 yellowbox1)
        (is_reachable shippingbox2 yellowbox2)
        (is_reachable yellowbox1 shippingbox1)
        (is_reachable yellowbox2 shippingbox2)

        (is_reachable shippingbox1 shippingbox2)
        (is_reachable shippingbox2 shippingbox1)

        (= (yellowbox-position left_nomansland) 1)
        (= (yellowbox-position yellowbox1) 2)
        (= (yellowbox-position yellowbox2) 3)
        (= (yellowbox-position yellowbox3) 4)
        (= (yellowbox-position yellowbox4) 5)
        (= (yellowbox-position yellowbox5) 6)
        (= (yellowbox-position right_nomansland) 7)

        (= (abs_X_position left_nomansland) 1)
        (= (abs_X_position yellowbox1) 2)
        (= (abs_X_position yellowbox2) 3)
        (= (abs_X_position yellowbox3) 4)
        (= (abs_X_position yellowbox4) 5)
        (= (abs_X_position yellowbox5) 6)
        (= (abs_X_position right_nomansland) 7)
        (= (abs_X_position shippingbox1) 1)
        (= (abs_X_position shippingbox2) 2)

        (= (abs_Y_position left_nomansland) 1)
        (= (abs_Y_position yellowbox1) 1)
        (= (abs_Y_position yellowbox2) 1)
        (= (abs_Y_position yellowbox3) 1)
        (= (abs_Y_position yellowbox4) 1)
        (= (abs_Y_position yellowbox5) 1)
        (= (abs_Y_position right_nomansland) 1)
        (= (abs_Y_position shippingbox1) 2)
        (= (abs_Y_position shippingbox2) 2)

        (is_reachable shippingbox1 shippingbox1)
        (is_reachable shippingbox2 shippingbox2)
        (is_reachable yellowbox1 yellowbox1)
        (is_reachable yellowbox2 yellowbox2)
        (is_reachable yellowbox3 yellowbox3)
        (is_reachable yellowbox4 yellowbox4)
        (is_reachable yellowbox5 yellowbox5)
        (is_in disk1 yellowbox1)
        (is_in disk2 yellowbox1)
        ...
        (is_in disk15 yellowbox5)

        (is_precise disk1 yellowbox1)
        ...
        (is_precise disk15 yellowbox5)
)
```

The goal is to put 6 parts in each shipping box and position them exactly where they were intended.

```
(:goal
        (and
        (is_in disk1 shippingbox1)
        (is_in disk2 shippingbox1)
        (is_in disk3 shippingbox1)
        (is_in disk4 shippingbox1)
        (is_in disk5 shippingbox1)
        (is_in disk6 shippingbox1)
        (is_in disk7 shippingbox2)
        (is_in disk8 shippingbox2)
        (is_in disk9 shippingbox2)
        (is_in disk10 shippingbox2)
        (is_in disk11 shippingbox2)
        (is_in disk12 shippingbox2)

        (is_precise disk1 shippingbox1)
        (is_precise disk2 shippingbox1)
        (is_precise disk3 shippingbox1)
        (is_precise disk4 shippingbox1)
        (is_precise disk5 shippingbox1)
        (is_precise disk6 shippingbox1)
        (is_precise disk7 shippingbox2)
        (is_precise disk8 shippingbox2)
        (is_precise disk9 shippingbox2)
        (is_precise disk10 shippingbox2)
        (is_precise disk11 shippingbox2)
        (is_precise disk12 shippingbox2)

        )
)
```

It is possible to set alternate metrics for what the planner should optimize, these settings have a huge impact on the time it takes to create a solution. By default, it optimizes for time for durative actions. For atomic actions, the goal is to reduce the number of actions.

```
(:metric minimize total-time )
```

4.1.3 Solution file

It depends on the planner how the solution file is created. But the planners I tried, used this format: action start time: (action name, parameters) [action duration] A solution made by LPG-td looks like this:

```
; Version LPG-td-1.4
; Seed 33248757
; Time 0.84
; Search time 0.38
; Parsing time 0.03
; Mutex time 0.43
; MetricValue 1925.00

0.0003:   (UR_MOVE UR UR_START YELLOWBOX5) [2.0000]
2.0005:   (UR_PICK UR YELLOWBOX5 DISK10) [5.0000]
0.0003:   (SHIPPINGBOX_MOVE SHIPPINGBOX1 CONV_LOCATION_BAD YELLOWBOX5) [2.0000]
7.0007:   (UR_PLACE_FAST UR DISK10 YELLOWBOX5 SHIPPINGBOX1) [5.0000]
12.0010:   (UR_PICK UR YELLOWBOX5 DISK8) [5.0000]
17.0012:   (UR_PLACE_FAST UR DISK8 YELLOWBOX5 SHIPPINGBOX1) [5.0000]
22.0015:   (UR_PICK UR YELLOWBOX5 DISK9) [5.0000]
27.0018:   (UR_PLACE_FAST UR DISK9 YELLOWBOX5 SHIPPINGBOX1) [5.0000]
32.0020:   (UR_PICK UR YELLOWBOX5 DISK6) [5.0000]
37.0023:   (UR_PLACE_FAST UR DISK6 YELLOWBOX5 SHIPPINGBOX1) [5.0000]
42.0025:   (UR_PICK UR YELLOWBOX5 DISK7) [5.0000]
...
```

The solution file is a list of actions that lead from the initial state to the goal.

4.2 Planner

The quality of the solution depends largely on the kind of planner we use. There are a large number of free to use planners available, but most are over 10+ years old, so it takes a lot of effort to compile and run them.

I tried several planners e.g. OPTIC, POPF, FF, Metric-FF, SMTPlan, but the one that best suited my planning problem and the PDDL language expressions I intended to use was LPG-td planner.

I chose the planner based on the fact that it should support at least the PDDL 2.1 language expressions, its output should suit to what ROSPlan accepts, and produce good quality solution files within the foreseeable future.

4.2.1 LPG-td

The LPG-td planner was chosen because it met my expectations in all aspects and in addition it is available as an executable.

LPG[36] (Local search for Planning Graphs) is a planner based on local search and planning graphs that handles PDDL 2.1 domains involving numerical quantities and durations. The system can solve both plan generation and plan adaptation problems. The basic search scheme of LPG was inspired by Walksat, an efficient procedure to solve SAT-problems. The search space of LPG consists of "action graphs", particular subgraphs of the planning graph representing partial plans. The search steps are certain graph modifications transforming an action graph into another one. LPG exploits a compact representation of the planning graph to define the search neighborhood and to evaluate its elements using a parametrized function, where the parameters weight different types of inconsistencies in the current partial plan, and are dynamically evaluated during search using discrete Lagrange multipliers.

The evaluation function uses some heuristics to estimate the "search cost" and the "execution cost" of achieving a (possibly numeric) precondition. Action durations and numerical quantities (e.g., fuel consumption) are represented in the actions graphs, and are modelled in the evaluation function. In temporal domains, actions are ordered using a "precedence graph" that is maintained during search, and that takes into account the mutex relations of the planning graph.

The system can produce good quality plans in terms of one or more criteria. This is achieved by an anytime process producing a sequence of plans, each of which is an improvement of the previous ones in terms of its quality. LPG is integrated with a best-first algorithm similar to the one used by FF. The system can automatically switch to best-first search after a certain number of search steps and "restarts" have been performed. Finally, LPG can be used as a preprocessor to produce a quasi-solution that is then repaired by ADJ, a plan-analysis technique for fast plan-adaptation.

LPG-td[37] is an extension of the LPG planner that can handle most of the features of PDDL 2.2, the standard planning language of the 4th International Planning Competition (IPC-4).In particular, LPG-td is an incremental fully-automated planner generating plans for problems in domains involving:

- STRIPS actions
- durative actions
- actions and goals involving numerical expressions
- operators with universally quantified effects
- operators with existentially quantified preconditions
- operators with disjunctive preconditions
- operators with implicative preconditions
- timed initial literals (deterministic unconditional exogenous events)

- predicates derived by domain axioms
- maximization or minimization of complex plan metrics

There are three ways to start the planner[36]:

- LPG-td.speed finds as quickly as possible a plan and then stops
- LPG-td.quality finds a plan and then spends a certain amount of CPU-time (automatically decided) trying to improve it
- LPG-td.bestquality incrementally finds the best plan that the planner can derive within an user-specified CPU-time limit

4.3 Changing the simulated environment

In this section, I summarize the changes I made to the ARIAC environment so that the world described in PDDL and the simulated world can be matched.

4.3.1 Changing the behaviour of the conveyor belt

In the original environment, the conveyor belt could only be moved in one direction. This greatly limits the number of environmental modifiers. Although, due to the properties of the planner, the planner is still planning so that the conveyor belt can only move in one direction, at least in the future it is possible to change this.

In order to change the direction of the conveyor belt, the ConveyorBeltPlugin had to be modified to accept negative values as a command, and to move the belt in the opposite direction to negative values.

Once the belt has travelled a certain distance, a new shipping box will teleport to the end of the belt. This is not a problem if the belt can only move in one direction, because then the boxes can't get jammed up, but I would have had a problem with the newly produced shipping boxes. To avoid this, a GEAR plugin had to be modified to prevent spawning new boxes.

4.3.2 Modification of the box model file

The distance between the centers of two yellow boxes is 0.81 centimetres. The goal is to have the centers of the two shipping boxes at such a distance on the assembly line, but this is not possible without modification. By removing the cover of the boxes, they can be placed at a sufficient distance from each other. This requires modifying the model files for the boxes.

4.3.3 Installation of sensors

In order to know where the shipping boxes are on the conveyor belt, it is necessary to install sensors. Each logical position (which in the solution file are yellowbox1-5, right / left_nomansland) has a physical counterpart, so to know that the boxes have reached the right place I placed break beams next to the belt.

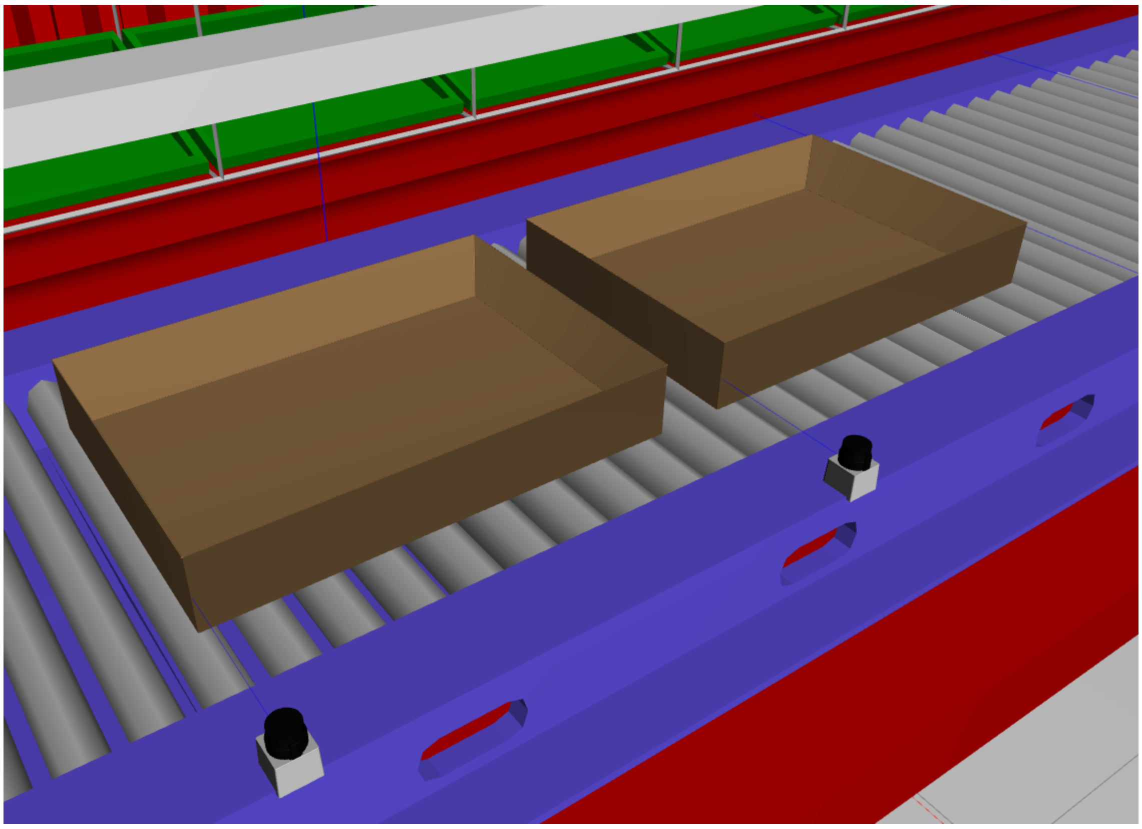


Figure 4.1: *The figure shows a completed order in the shipping boxes. Next to the conveyor belt are the break beam sensors with their beams displayed.*

4.4 Modifying and using ROSPlan

The ROSPlan package provides the connection between ROS and AI planning. The 3.4 figure shows that the developer only needs to do low level actions because the other overheads are handled by ROSPLan. Unfortunately, this was not the case, because I had some code from my previous projects that I wanted to keep and it was written in Python and ROSPlan is in C++. Therefore, some interfaces had to be created, which creates a connection between python and C++.

Although ROSPlan supports LPG-td planner with an interface, this interface when

I started the development was not perfect and did not work. The problem was that the planner creates the solution with all uppercase characters and the interface only accepts lowercase characters. The other problem was that the planner does not list the execution time of each action in chronological order when creating the solution file. Therefore, each character had to be converted to lowercase letters and the actions had to be chronologically ordered.

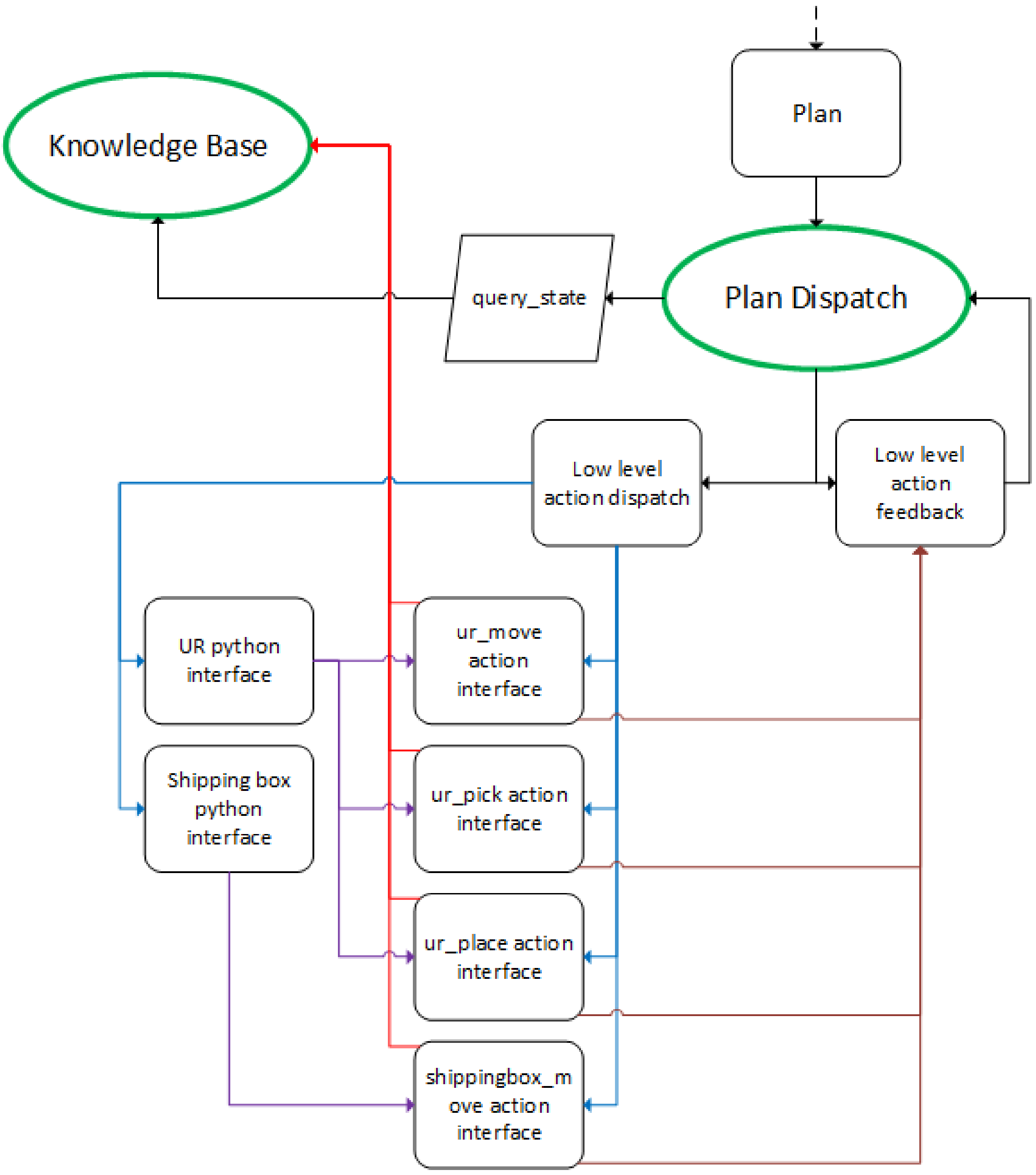
I created two interfaces, one for the UR robot arm and one for moving the shipping boxes. These of course does not obviate the need for ROSPlan action interfaces, which are responsible for updating the knowledge base. The plan dispatch node publishes the action to be executed on the action_dispatch topic. The Python and action interfaces are subscribed to this topic. If the interfaces notice that the published message is addressed to them, they will execute their task in parallel. When the Python interface indicates to the action interface that the action has been completed, the action interface will update the knowledge base with the effects. This system is shown in the 4.2 figure.

4.4.1 Creating low-level actions

When I started creating low-level actions, I did not have to start from scratch, because I already had code for controlling the UR robot arm in Cartesian space from previous projects. The parts are always in the same pose, so I linked the name and location of the parts, this way there is no need to use a camera to locate the parts.

The robot arm picks up the parts with a simulated pneumatic gripper. This can be remotely controlled by a ROS service. An object will be attached to the gripper if they are making contact. The gripper regularly publishes its status. The published message contains whether the suction is enabled or disabled or whether there is an object attached to the gripper.

The order served by the robot arm contains the number of parts and the pose to be placed in the shipping box. This order is published on a ROS topic at the start of the simulation. The position of the box is required for the exact positioning of all items. This would require additional sensors. For the sake of simplicity, I have not chosen this solution. Gazebo has a topic that can be used to change the pose of components in the environment. By selecting the reference frame for the right shipping box, the part can be teleported to its final position without the need for additional sensors for positioning. For the sake of realism, the robot arm turns to place the part in the box, but only to drop the part into the box.



Knowledge Base
Plan
Plan Dispatch
query_state
Low level action dispatch
Low level action feedback
UR python interface
Shipping box python interface
ur_move action interface
ur_pick action interface
ur_place action interface
shippingbox_move action interface

Figure 4.2: *The figure shows the connection between the interfaces. The red arrows indicate the service responsible for updating the knowledge base. The blue arrows indicate the topic on which the actions to be executed are published. Purple arrows indicate the topic through which the python interface indicates to the action interface that the action is complete. The brown arrows indicate the action feedback to the dispatcher.*

Chapter 5

Results and evaluation

5.1 Issues with planning

When creating PDDL files that describe the physical world, it does matter how much the planner is limited. If the state space is too large, it will take a long time to get a good quality solution. If the metrics on which we want to get the optimal plan are too complicated, the planning time will increase. Many attempts are needed to create files that model the environment well and produce optimal or near-optimal solutions.

The issues with classical planning are that the environment is treated as static and fully observable. In my work, it is no problem that the environment is treated as static, since it is a closed robot cell, thus something changes only if the agent intervenes. However in an application where the robot collaborates with humans, assuming that the environment is static is not correct. Not treating the environment as fully observable is a long-researched problem for AI planning researchers. [38] A good solution to this problem is that if something proves not to be what we had previously assumed, we will make a new plan using the new information. In order not to drastically increase the time spent on solution search and optimization, it is faster to start from the previous solution and repairing it, than replanning from scratch. [39] In my work, replan occurs when a low level action fails to execute. In this case, a new problem file is generated from the current state of the world and a new solution is created using it.

With the LPG-td planner, a lot depends on how good the quality of the first solution is, because it uses it to create a better optimized solution. It uses a random seed to create the first solution, so each run produces a different solution. That is why, although it strives to find the optimum, different runs result in different scores. In order to get the number of iterations to find a near-optimal solution, I created a script that calls the planner 10 times to optimize the first solution 10 times during each run. I set a time limit of 10 minutes for each run.

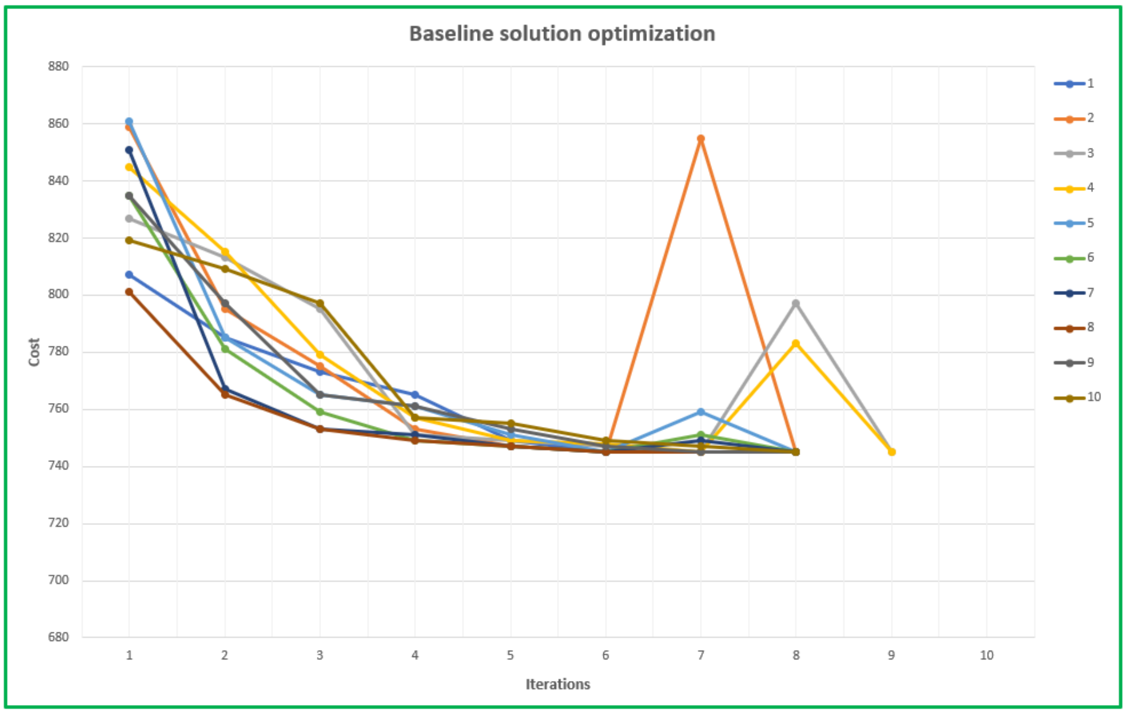


Figure 5.1: *The figure shows the results of the tests. The results of each run converge towards the optimum.*

In the 5.1 figure I summarized the results of the tests. A total of 80 measurements were made because the planner could not produce 10 measurements in each iteration in 10 minutes. It is clear from the diagram that the quality of the first solution is poor for each run, but the 4-5 iterations are very close to the optimum. The diagram also shows that each run eventually converges to nearly the same value.

The three outliers at the end of the run 2, 3, 4 are because the planner had not been able to optimize the existing solution for a long time, thus it started with a new solution that performed worse, so it returned to the previous solution.

5.2 Visualization of the plan

In order to keep track of what are the active actions during the simulation and to visually see the solution, a script generates an image from the solution file. The 5.2 figure shows a solution visually depicted. Each arrow is a pick or place action. There are places on the vertical axis and time on the horizontal axis. The red line indicates the movement of the UR robot arm. The gray and black lines indicate the movement of the shipping box. During the simulation, each time a new action is active, the image is regenerated, showing the currently active action. A textual solution file is more difficult for people to process, but it is much easier to notice in an image if the planner is not generating a plan like we

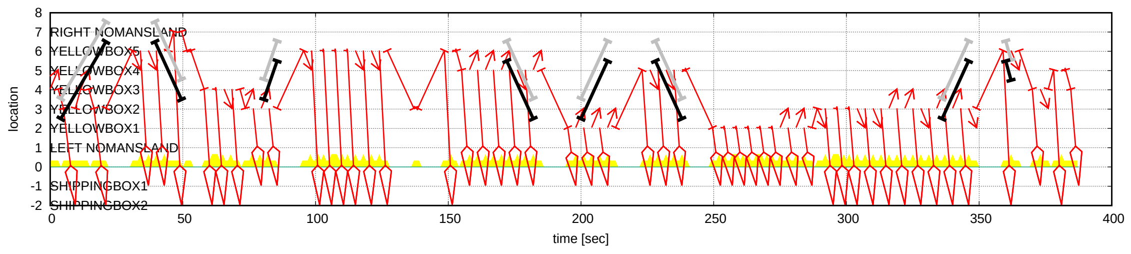


Figure 5.2: *The figure shows a solution file visually.*

thought.

The 5.2 figure in yellow shows how many actions are performed in parallel. In this environment, only one robot does all the tasks, but the goal is to complement the environment with more robots that cooperate.

When the graph is regenerated, the horizontal axis shows the simulation time, not the time it was in the solution file. This is an easy way to evaluate the difference between the PDDL model and reality. With the values specified in my problem file, the simulation time to take an action and the time that the planner assumes to execute an action are in the same order of magnitude. For example, the time required to add parts is not significantly different, so the values obtained in the simulation can be put back into the planning model. This also applies to the other actions. Where it is not possible to improve the relationship between the simulated and modelled values by correctly selecting the X, Y distances, it can be improved by correctly selecting the final multiplication factor.

5.3 Advantages of the simulation

The biggest advantage of using simulation is that it is not necessary to build a robot cell in order to obtain data about its efficiency or other characteristics. During the ARIAC, teams are also scored based on built-in metrics defined in the ARIAC scoring. Using these metrics, we can also get useful information about the goodness of a particular solution.

The ARIAC Key Performance Indicators (KPIs) are received when the completed order is shipped. The quality of an execution is described by the following factors:

- Total game score: the sum of each order's Completion Score.
- Total process time: the time it takes to send out the last order
- Product travel time: the time spent by the parts attached to the vacuum gripper

Shipment Completion Score [40] is the sum of:

- Product presence: One point for each correct product in the shipping box.

- All products: Points totaling the number of products, if and only if all products are in the shipping box and no additional unwanted/faulty products are in the shipping box.
- Product pose: One point for each product that is in the correct pose (location and orientation) in the shipping box. The location must be within 3cm of the target and the orientation must be within 0.1 radians.

For example, a solution received the following score:

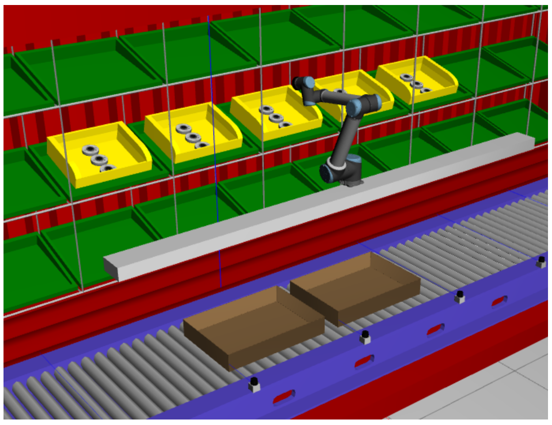
```
Score breakdown:
<game_score>
Total game score: [36]
Total process time: [176.509]
Product travel time: [61.515]
<order_score order_0>
Total order score: [18]
Time taken: [157.004]
Complete: [true]
<shipment_score order_0_shipment_0>
Completion score: [18]
Complete: [true]
Submitted: [true]
Product presence score: [6]
All products bonus: [6]
Product pose score: [6]
</shipment_score>

</order_score>

<order_score order_1>
Total order score: [18]
Time taken: [173.507]
Complete: [true]
<shipment_score order_1_shipment_0>
Completion score: [18]
Complete: [true]
Submitted: [true]
Product presence score: [6]
All products bonus: [6]
Product pose score: [6]
</shipment_score>

</order_score>

</game_score>
```

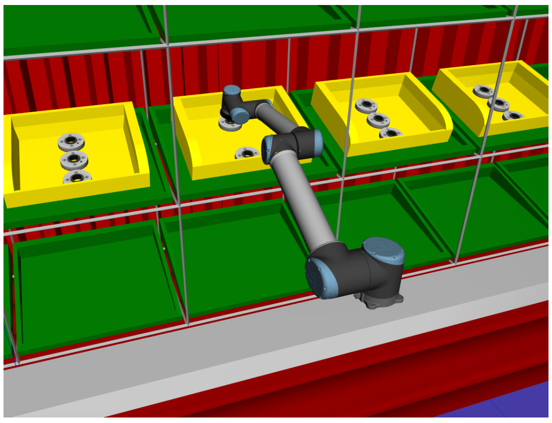
Total process time is a great indicator of how good the quality of a solution is. We can also compare two solutions based on how different the part travel time is. In a solution where the robot arm moves parts too much from one container to another, much can be accelerated. It is very difficult to give mathematical formulas for estimating robot cell service times when there are unforeseen events in the system. But based on the data taken from the simulated robot cell, we can give a good estimate. With AI planning there are many solutions that can be simulated and compared with results to evaluate the properties of a robot cell, much quicker than describing them with mathematical models. Another advantage is that mathematical models may not remain valid in case of changes, but PDDL files can be modified quickly.

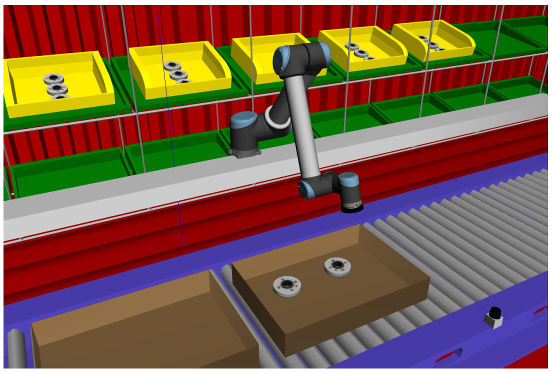
5.4 Executing the solution

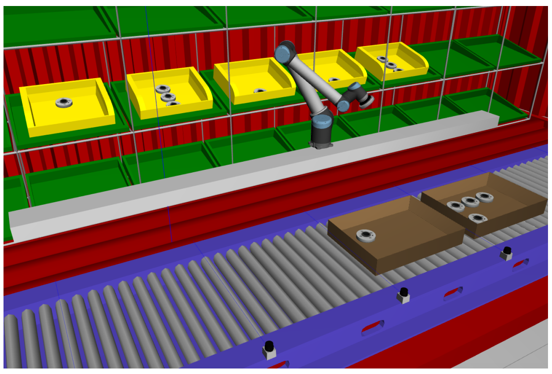
The 5.3 figure shows the robot arm fulfilling two orders. The order parameters are received via a ROS topic, which includes the type of parts and where to place them in the shipping box. The robot does not follow a pre-programmed choreography, so it can easily adapt to changes. ROSPlan uses a structure based on interfaces, so it is easy to replace the planner or other components with another one if it performs better. The robot arm has three actions and the conveyor belt can be controlled with a single command. For the drone that delivers the finished orders does not need to write an action, because it does not change the state of the environment, just to get the ARIAC scores.

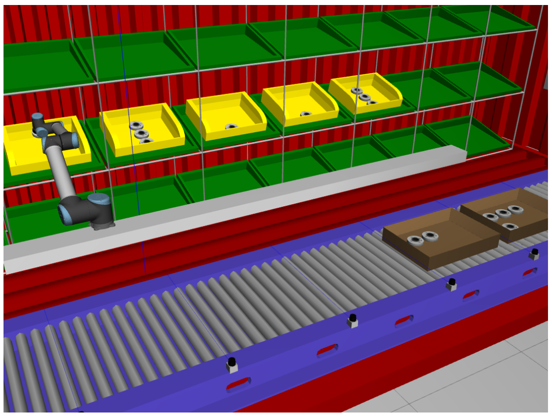
The use of artificial intelligence has never occurred in the history of ARIAC so far. However, the need for an entity that plans to fulfil orders has always been present. Nonetheless, modifications are needed to be able to be used in the real competition. For example, the actions on the robot arm would have to be completed, because the placement of parts with teleportation is not appropriate. The simulation revealed that this is a valid approach to a task that is present in the industry and provides a workable solution.


(a)


(b)


(c)


(d)


(e)

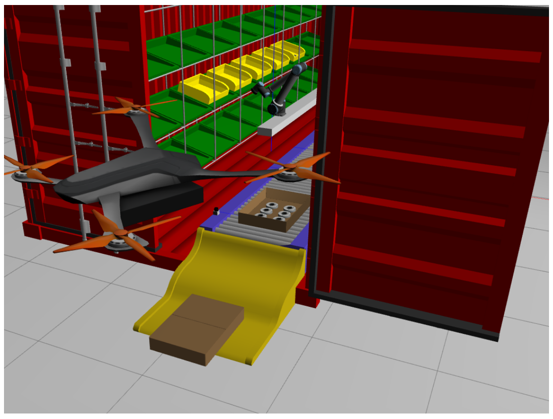

(f)

Figure 5.3: *The pictures show important moments of a solution execution, from empty shipping boxes to completed orders*

Chapter 6

Conclusion and further work

This work presents a possible approach for making industrial robotic cells more adaptable. For this, I used AI planning to solve a common task in the industry. AI planning is a long-established branch of research in artificial intelligence. AI planning is not used to solve real-world tasks because it requires assumptions about the world that are rarely found in reality. To overcome these preconditions, I presented a possible approach in this paper. My goal was to make this approach applicable to other types of tasks.

For the simulation I chose the 2019 ARIAC environment. The task can be summarized as a robot arm fulfils orders by placing the desired parts in a shipping box on a conveyor belt. For the robot arm, AI provides a plan to complete the order. When creating a plan, the planner takes into account environmental variables and optimizes the solution based on a predefined metric. The connection between the simulated environment and the AI is provided by the ROSPlan framework. The information obtained from the simulation can be used to optimize a robot cell and to estimate the average serving time for a robot cell.

As a next step, I would like to perform tests to check the improvement of the ARIAC KPIs with multiple robotic arms. With different types of robotic arms, it may be interesting to consider other metrics such as power consumption or radio spectrum utilization for remote control.

It might be worthwhile to try AI planning for another industrial problem to see how it could be improved. Such problems, may be the use of service robots or the control of robots where goods have to be transported from one workstation to another.

Bibliography

[1] Laurence Cruz. Digitization and IoT reduce production downtime. https://newsroom.cisco.com/feature-content?type=webcontent&articleId=1764957, September 2019.

[2] Jim Schmidt Ron Harbour. Tomorrow's Factories Will Need Better Processes, Not Just Better Robots. https://hbr.org/2018/05/tomorrows-factories-will-need-better-processes-not-just-better-robots, September 2019.

[3] Siemens Plant Simulation. https://www.plm.automation.siemens.com/store/en-us/plant-simulation/, December 2018.

[4] Gazebo Robot Simulator. http://gazebosim.org, December 2018.

[5] Agile Robotics for Industrial Automation Competition (ARIAC). http://gazebosim.org/ariac, December 2018.

[6] Introduction to ROS, ROS Wiki. http://wiki.ros.org/ROS/Introduction, December 2018.

[7] Solange Lemai and François Ingrand. Interleaving temporal planning and execution in robotics domains. pages 617–622, 01 2004.

[8] M. Cashmore, M. Fox, D. Long, D. Magazzeni, and B. Ridder. Opportunistic planning in autonomous underwater missions. *IEEE Transactions on Automation Science and Engineering*, 15(2):519–530, April 2018.

[9] T. Estlin, D. Gaines, C. Chouinard, R. Castano, B. Bornstein, M. Judd, I. Nesnas, and R. Anderson. Increased mars rover autonomy using ai planning, scheduling and execution. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4911–4918, April 2007.

[10] D. E. Bernard, G. A. Dorais, C. Fry, E. B. Gamble, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. P. Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, and B. C.

Williams. Design of the remote agent experiment for spacecraft autonomy. In *1998 IEEE Aerospace Conference Proceedings (Cat. No.98TH8339)*, volume 2, pages 259–281 vol.2, March 1998.

[11] Ratiu, Mariana and Adriana Prichici, Mariana. Industrial robot trajectory optimization- a review. *MATEC Web Conf.*, 126:02005, 2017.

[12] Ron Alterovitz, Sven Koenig, and Maxim Likhachev. Robot planning in the real world: Research challenges and opportunities. *AI Magazine*, 37(2):76–84, Jul. 2016.

[13] Lentin Joseph. *Mastering ROS for Robotics Programming*. Packt Publishing, December 2015.

[14] Is ROS for me? `http://www.ros.org/is-ros-for-me/`, December 2018.

[15] ROS Concepts, ROS Wiki. `http://wiki.ros.org/ROS/Concepts`, December 2018.

[16] ROS Packages, ROS Docummentation. `http://docs.ros.org/independent/api/rospkg/html/packages.html`, December 2018.

[17] ROS Nodes, ROS Wiki. `http://wiki.ros.org/Nodes`, December 2018.

[18] ROS Technical Overview, ROS Wiki. `http://wiki.ros.org/ROS/TechnicalOverview`, December 2018.

[19] ROS Master, ROS Wiki. `http://wiki.ros.org/Master`, December 2018.

[20] ROS Messages, ROS Wiki. `http://wiki.ros.org/Messages`, December 2018.

[21] ROS Standard messages, ROS Wiki. `http://wiki.ros.org/std_msgs`, December 2018.

[22] ROS Common Messages, ROS Wiki. `http://wiki.ros.org/common_msgs`, December 2018.

[23] ROS Topics, ROS Wiki. `http://wiki.ros.org/Topics`, December 2018.

[24] GEAR interface used in the ARIAC 2018. `https://bitbucket.org/osrf/ariac/wiki/2018/competition_interface_documentation`, December 2018.

[25] ROSPlan documentation. `https://kcl-planning.github.io/ROSPlan/documentation/`, October 2019.

[26] Open dynamics engine website. `http://www.ode.org/`, December 2018.

[27] Open dynamics engine manual. `https://www.ode-wiki.org/wiki/index.php?title=Manual:_All&printable=yes`, December 2018.

[28] Nathan Koenig and Andrew Howard. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, page 2149.

[29] Gazebo components. `http://gazebosim.org/tutorials?tut=components`, December 2018.

[30] Gazebo ROS Integration. `http://gazebosim.org/tutorials?tut=ros_overview`, December 2018.

[31] UR10 Robot Arm. `https://www.universal-robots.com/products/ur10-robot/`, December 2018.

[32] Joerg Hoffmann. *Everything You Always Wanted to Know About Planning (But Were Afraid to Ask).* 34th Annual German Conference on Artificial Intelligence - KI'11, Oct 2011, Berlin, Germany. Springer, 7006, pp.1-13, 2011, Lecture Notes in Artificial Intelligence.

[33] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[34] PDDL overview. `http://csci431.artifice.cc/notes/pddl.html`, September 2019.

[35] AI planning. `https://researcher.watson.ibm.com/researcher/view_group.php?id=8432`, September 2019.

[36] Lpg-td homepage. `http://zeus.ing.unibs.it/lpg/`, September 2019.

[37] Ivan Serina Paolo Toninelli Alfonso Gerevini, Alessandro Saetti. Lpg-td: a fully automated planner for pddl2.2 domains. 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04), 2004.

[38] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong planning under partial observability. *Artificial Intelligence*, 170:337–384, 04 2006.

[39] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. volume 2006, pages 212–221, 01 2006.

[40] ARIAC Scoring. `https://bitbucket.org/osrf/ariac/wiki/2018/scoring`, September 2019.